Tracing the Rationale Behind UML Model Change Through Argumentation

Ivan J. Jureta and Stéphane Faulkner

Information Management Research Unit (IMRU), University of Namur, Belgium
iju@info.fundp.ac.be, stephane.faulkner@fundp.ac.be

Abstract. Neglecting traceability—i.e., the ability to describe and follow the life of a requirement—is known to entail misunderstanding and miscommunication, leading to the engineering of poor quality systems. Following the simple principles that (a) changes to UML model instances ought be justified to the stakeholders, (b) justification should proceed in a structured manner to ensure rigor in discussions, critique, and revisions of model instances, and (c) the concept of argument instantiated in a justification process ought to be well defined and understood, the present paper introduces the UML Traceability through Argumentation Method (UML-TAM) to enable the traceability of design rationale in UML while allowing the appropriateness of model changes to be checked by analysis of the structure of the arguments provided to justify such changes.

1 Introduction

In a noted discussion of the traceability problem [10], Gotel and Finkelstein define traceability as follows:

> "Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through all periods of on-going refinement and iteration in any of these phases)."

Ensuring proper traceability through specialized concepts, techniques, and methods is argued to reduce the number of iterations in the construction and change of requirements engineering (RE) artifacts, thus helping keep the software development project under time, budget, and other constraints. However, if traceability is neglected, misunderstanding and miscommunication are bound to appear, compounding the loss of implicit information guiding requirements change and increasing the risk of poor project results [6,20,25].

This paper focuses on the problem of tracing the rationale behind changes local to one or spanning across several different kinds of models in the Unified Modeling Language (UML) [18]. To address the problem, the UML Traceability through Argumentation Method (UML-TAM) is suggested to enable the traceability of design rationale in UML while allowing the appropriateness of model changes to be checked by analysis of the structure of the arguments provided to justify such changes.

As the related research efforts are numerous, the following section (§2) first positions the present work within the relevant literature. The problem of interest is then identified and contributions outlined (§3), and is followed by a description of the case study (§4). The conceptual basis of UML-TAM is then presented (§5). It is followed by an illustration of its use in the case study (§6). The paper closes with conclusions and indications on directions for future effort (§7).

2 Background and Related Work

Complexity of the traceability problem, its span over the various activities in software development, along with the trade-off between extensive traceability and budget and time constraints make elusive the construction of an encompassing traceability approach still applicable to realistic settings—methods specialized for particular traceability sub-problems, combined with domain-specific expertise on when and how to apply them in a given project seem to be the choice in research and industry. In light of the various methods suggested in related research efforts, situating the results of the present paper is facilitated by classification over five taxonomic dimensions: traceability data types, scope, degree of automation, conceptual foundations, and framework specificity. Each is considered in turn below.

2.1 Traceability Data Types

Traceability data types, as suggested by Dömges and Pohl [6], distinguish methods according to the content of traceability information being recorded:

– *Bi-directional links* between the stakeholder expectations, derived requirements, and software components enable validation of system functionality by stakeholders and impact analysis of requirements change on the system. Ramesh and colleagues [25] indicate that such benefits can be achieved, although at high initial cost of implementing and applying traceability policies. A framework allowing the capture of bi-directional links has been proposed by Pohl [21] and later extended to allow configuration to project-specific traceability needs [22], in both cases focusing on the recording of what changes are made, by whom, when, and how.
– *Contribution structures* aim at clearly relating the requirements to stakeholders to facilitate negotiation, search for additional information, and revision. Gotel [11] introduced contribution structures in RE to allow the recording of detailed information on stakeholders and the requirements they provide, hence ensuring traceability of the requirements to the people and systems from which these emanate.
– *Design rationale* records the reasoning that led to particular modeling and other software development decisions, in the aim of arriving at a shared understanding of models and other artifacts, and their purpose in the given

project. Usually, a design rationale approach is employed to record such traceability information (Louridas and Loucopoulos give an overview [16]).
– *Process data* which relates to the planning and control of activities in the software development project.

2.2 Scope

Gotel and Finkelstein [10] introduce a separation of pre-Requirements Specification (pre-RS) from post-RS traceability. *Pre-RS*, which concerns the life of stakeholder expectations until they are converted to requirements, has been treated in the various RE frameworks proposed over the last decade—for instance, the introduction of goals in requirements models facilitates traceability, for goals make explicit (at least in part) the rationale for the inclusion of more specific requirements [30]. *Post-RS* focuses on the evolution of requirements in the steps following RE, i.e., the various activities involved in deploying the requirements. Automated traceability methods (below) focus on post-RS.

2.3 Degree of Automation

The degree of automation concerns the support allowed by or provided with a traceability method to reduce manual effort and facilitate analyses of trace information. Haumer and colleagues [12] and Jackson [13] both suggest manual traceability techniques focused on simplicity, while allowing rich trace recording (e.g., video, audio, etc.). Such an approach becomes difficult to manage efficiently for realistic systems, leading to, among other, Egyed's proposal [7] where models and software are aligned using traces generated by observation of software operation through the running of various test scenarios. Antoniol and colleagues [1] and Pinheiro and Goguen [19] both rely on formal methods for traceability, with the difficulty of avoiding obsolescence of formal trace specifications.

2.4 Conceptual Foundations

Conceptual foundations discriminate according to the main concepts employed in recording traceability information (e.g., goals, scenarios, aspects). Egyed [7] generates design traceability information by iteratively running test *scenarios* on already operational software, so as to verify whether the models implementing the tested functionality correspond to the behavior of the observed system. A preliminary proposal from Naslavsky and colleagues [17] focuses on traceability between scenarios and the use thereof to relate requirements to code. Ubayashi and colleagues [28] propose a method for dealing with model evolution using model transformations based on *aspect* orientation, the main benefit thereof being the separation of concerns over traceability information. Torenzo and Castro [27] also seem to separate concerns, albeit through specialized *views* and not aspects. In an overview of goal-oriented RE [30], Van Lamsweerde observes that the refinement links in goal refinement trees, in which an abstract goal is made more precise through refinement, can be read as traceability links making *goal*-orientation a favorable approach to aligning abstract and precise, operational information about a system. The concept of *argument* appears in design rationale approaches (for an overview, see [16]) which enable the recording of reasoning behind decisions. For instance, Ramesh and Dhar [24] suggest an approach involving concepts specialized for the RE: in addition to classical concepts—position, argument, issue—introduced in IBIS [5], REMAP [24] integrates the notion of requirement, design object, decision, and constraint.

2.5 Framework Specificity

Framework specificity classifies approaches according to whether they are specialized or not for a particular software development framework. Briand and colleagues [3] suggest bi-directional links be extracted automatically from changes in UML models, whereby each identified type of UML model refinement (each refinement being a kind of model change) has associated traceability information, thus facilitating impact analysis in model evolution. Letelier [15] suggests a roughly defined metamodel of traceability information to collect when working with UML and requirements expressed in textual form; the aim is to ensure that bi-directional links are known during UML modeling, while very limited support is provided for design rationale recording.

3 Problem Outline and Contributions

The work presented in the remainder enables the recording of design rationale behind changes local to one or spanning across several different kinds of UML models. It is thus framework-specific and both pre- and post-RS (this depending on how UML is employed), while relying on the concept of argument. Because informally or formally expressed information is allowed into arguments to allow adaptability of the method to project specificities, automation is limited, this entailing selective application of the method. The present work is a response to the following observations, each highlighting a difficulty in current research:

- UML traceability rarely aims to record the rationale behind modeling decisions, and when this is attempted, as in Letelier's work [15], very limited attention is given to what kind of rationale information is to be recorded and how, and if/how it can be analyzed.
- Automated traceability by taxonomies of UML change/refinement types lacks the recording of design rationale—in the efforts cited in §2, traceability information answers *what* changes are made, but not *why* they are made. It is therefore possible to determine who, when, and how made a particular appropriate or inappropriate decision, but it is difficult/impossible to determine why the decision is made, hence limiting the potential to learn from mistakes or reinforce appropriate modeling behavior.
- Framework-independent traceability methods that use arguments in recording design rationale, such as REMAP [24] only provide techniques for trace capture—how to analyze such information remains unknown.

Following the simple principles that (a) changes to UML model instances ought be justified to the stakeholders, (b) justification should proceed in a structured manner to ensure rigor in discussions, critique, and revisions of model instances, and (c) the concept of argument instantiated in a justification process ought to be well defined and understood, the present paper introduces the UML Traceability through Argumentation Method (UML-TAM) for capturing and analyzing design rationale in UML modeling. The salient properties of the method are:

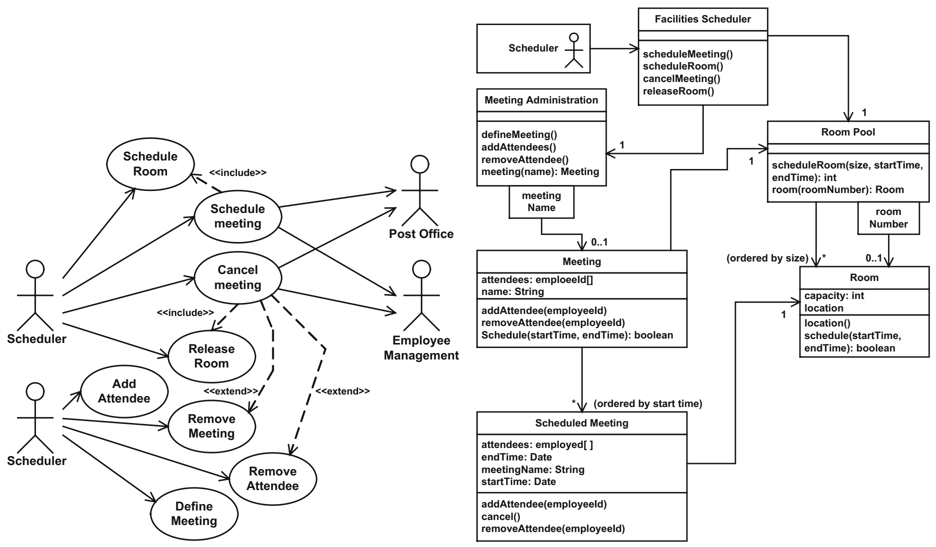
– *Adaptability.* Both informal and formal, and qualitative and quantitative information is allowed into arguments, to ensure that few constraints are placed on the stakeholders employing it to record design rationale.
– *Active rationale analysis.* Where available methods focus on ensuring design rationale is recorded (passive rationale traceability), UML-TAM provides specialized analyses for confronting arguments and avoiding ill-structured rationale which unavoidably leads to inappropriate modeling choices.
– *Sound conceptual foundations.* By relying on formal definitions of the concept of argument established in AI, and using it as a central concept, UML-TAM avoids ambiguity and aims to facilitate the learning of the method to the stakeholders (it merely requires the understanding of the notion of argument and the argumentation and justification processes).
– *Justification of modeling choices.* While recording arguments is certainly relevant, confronting them through a justification process to discriminate among alternative changes of model instances is critical. Justification thus provides a means for selecting among alternative sets of arguments to arrive at justifiably appropriate modeling choices.

4 Case Study

Following the classical meeting scheduler case study [29], a variant serves herein to illustrate the salient features of the method.[1] The aim is to design a system for scheduling meetings and meeting rooms. A user can request a meeting room of a chosen size and for a chosen period of time, and can schedule a meeting. A user can cancel any of the mentioned two until the beginning of the meeting time. An email is sent to participants any time the meeting is scheduled or canceled. When defining a meeting, the user provides a list of attendees, meeting time and room, and gives a brief description of the topic. It is further assumed that there is a Post Office package which delivers messages to designated users, and an Employee Management package which provides employee reference and email address. Fig.1(a) shows the initial use case which represents most of the described

[1] As noted above, UML-TAM is not intended for recording rationale behind all modeling decisions for it is not automated and thus impractical—contributions are primarily conceptual and not related to efficiency per se in the present paper. An accessible case study, appropriate for the constraints of the present format, thus introduces the method, while scalability and cost to industrial projects are under study.

functionality but is incomplete and serves as a starting point in moving toward a more extensive use case diagram to illustrate the use of UML-TAM in tracing rationale for change. Fig.1(b) gives an initial class diagram, and is used in the remainder to illustrate traceability within class diagram with UML-TAM.



(a) An initial use case diagram (b) An initial class diagram

Fig. 1. Some initial UML diagrams for the case study

5 Traceability Through Argumentation in UML-TAM

Returning to the initial use case digram in Fig.1(a), it is not difficult to notice it is incomplete at least with regards to the following plausible situations:

– If the room of requested size is not available at the requested period, various alternative responses by the system can be identified: e.g., it may record a failed request for a room for statistics on room availability; another option is to communicate the unavailability to the user and ask for a different period.
– If an attendee is added as a participant to a recurrent meeting, should the system assume that this person is to attend all occurrences of the meeting in the future, or should the user specify this? Same applies when an attendee of a recurrent meeting is removed from the list of participants—does the removal apply for all occurrences of the meeting or only the next one?
– A participant informed of a meeting may have another engagement for the same period. Fig.1(a) gives no explicit mechanisms for ensuring the scheduler knows what participants to expect. The system could, e.g., connect to employees' electronic agendas and return availability information when the scheduler adds a participant and the meeting period.

It should be apparent from the above that providing a revised use case diagram alone—i.e., without additional information on why that particular revision is more adequate than another one—may be appropriate only in case the stakeholders are of similar background, share a precise idea of what the system is expected to do, and so on. In most realistic settings, however, this is not satisfactory, for various stakeholders would participate, each bringing a different perspective on the system grounded in different backgrounds and interests. The very presence of alternatives in both system functionality and of options in the representation of functionality (e.g., at the level of use cases: what to wrap in an existing use case, what requires an additional use case, and so on) makes it appropriate to make explicit the reasons (i.e., arguments) that aim to justify the functionality and representation decisions. One thus observes that three components are needed for ensuring traceability of rationale in UML: (1) a design rationale approach (below: TAM-Design Rationale, TAM-DR), which indicates when and how the engineer proceeds to making explicit the alternatives in functionality and/or modeling; (2) an argumentation framework, which, as soon as the alternatives are known, enables the argumentation of each alternative, the confrontation and comparison of arguments, ending in a justified choice of one alternative; (3) specialized means for connecting the content of UML diagrams with the content of rationale traces (referred to in the remainder as TAM-Connectors) produced through the use of the design rationale approach and associated argumentation and justification techniques.

5.1 UML-TAM Design Rationale

Having identified an engineering problem, design rationale literature (and as usual in problem solving) suggests the engineer should identify alternative solutions, compare them according to some relevant criteria, subsequently choose one alternative, and act upon the prescription given in the alternative. In the classical IBIS approach [5], the aforementioned problem is termed *issue* whereas *positions* (i.e., alternative solutions) resolve issues, and *arguments* support or object to positions. A problem in the present setting appears whenever alternative system structures can be chosen to translate stakeholder expectations into a UML representation, or when several modeling options exist for a chosen alternative system structure (i.e., one knows what to model, but syntax and semantics of the model permit various ways of modeling this). Based on work from Louridas and Loucopoulos [16], which integrates common characteristics of established design rationale approaches, a design rationale approach specialized for rationale traceability in UML-TAM involves the following steps (see, Fig.2):

1. *Problem setting* consists of identifying a discrepancy between the content of the given UML model instance and the content it should represent—e.g., some newly acquired information is not represented therein, or the given representation uses questionable modeling choices.
2. Based on the problem statement produced in 1 above, *problem analysis* leads to the identification of *alternative solutions.*

3. *Evaluation* then consists of providing arguments for or against each alternative solution. Such *argumentation* is followed by a *justification* of a choice of (i.e., *Decision* on) a particular alternative.
4. Having selected the alternative, the affected UML model instances need to be changed according to the adopted solution. The process is reinitiated as new problems are identified.

As shown in Fig.2, content of alternatives and arguments can give itself rise to new problem statements. Activities of the given process rely mainly on the domain- and problem-specific knowledge of the stakeholders. Argumentation and justification activities require specialized concepts and techniques outlined in §5.2 and §5.3. The use of the given concepts and techniques is exemplified in §6.

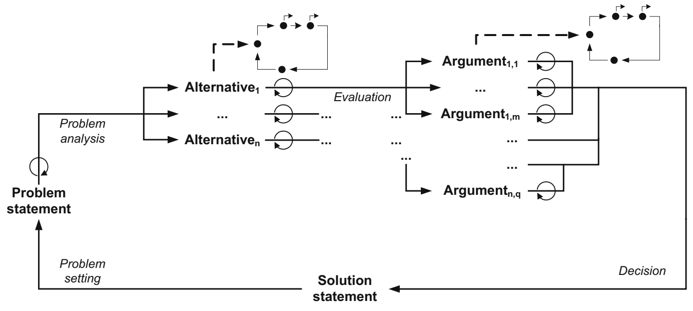


Fig. 2. Overview of the UML-TAM design rationale process

5.2 UML-TAM Argumentation Framework

Argumentation modeling literature [4] in the artificial intelligence field focuses on formalizing commonsense reasoning in the aim of automation. An *argumentation model* is a static representation of an *argumentation process*, which can be seen as a search for arguments, where an argument consists of a set of rules chained to reach a conclusion. Each rule can be rebutted by another rule based on new information. To formalize such defeasible reasoning, elaborate syntax and semantics have been developed (e.g., [4,26,2]) commonly involving a logic to formally represent the argumentation process and reason about argument interaction. A structured argumentation framework (i.e., a model and processes employing the model) is needed herein for a rigorous justification process in the *Evaluation* step of TAM-DR. To arrive at a structured argumentation system, the concept of argument is first defined below, followed by a set of argument relationships, and the justification process.

Argument. Assuming a first-order language $\mathcal{L}$ defined as usual, let $\mathcal{K}$ be a consistent set of formulae (i.e., $\mathcal{K} \nvdash \perp$), each a piece of information, and let $\mathcal{K} \equiv \mathcal{K}_N \cup \mathcal{K}_C$. Members of the set $\mathcal{K}_N$, called *necessary knowledge*, represent facts about the universe of discourse and are taken to be formulae which contain variables. Necessary knowledge is assumed unquestionable. The set $\mathcal{K}_C$, called *contingent knowledge*, are information that can be put in question or argued for. It is then said that the knowledge a stakeholder a can use in argumentation is given by the pair (K_a, Δ_a), where K_a is a consistent subset of $\mathcal{K}$ (i.e., $K_a \subset \mathcal{K}$ and $K_a \nvdash \bot$), and Δ_a is a finite set of *defeasible rules* of the form $\alpha \hookrightarrow \beta$. The relation $\hookrightarrow$ between formulae α and β is understood to express that "reasons to believe in the antecedent α provide reasons to believe in the consequent β". In short, $\alpha \hookrightarrow \beta$ reads "α is reason for β".

Let A a set of stakeholders, $K \equiv \bigcup_{a \in A} K_a$, and $\Delta \equiv \bigcup_{a \in A} \Delta_a$. Given (K_a, Δ_a) and $P \subset \Delta_a^{\downarrow}$, where $\Delta_a^{\downarrow}$ is a set of formulae from Δ_a instantiated over constants of the formal language, P is an *argument* for $c \in \mathcal{K}_C$, denoted $\langle P, c \rangle_K$, if and only if: 1) $K \cup P \mid\!\sim c$ (K and P derive c); 2) $K \cup P \nvdash \bot$ (K and P are consistent); and 3) $\nexists P' \subset P, K \cup P' \mid\!\sim c$ (P is minimal for K). Where "$\mid\!\sim$" is called the *defeasible consequence* [26] and is defined as follows. Define $\Phi = \{\phi_1, \ldots, \phi_n\}$ such that for any $\phi_i \in \Phi$, $\phi_i \in K \cup \Delta^{\downarrow}$. A formula ϕ is a defeasible consequence of Φ (i.e., $\Phi \mid\!\sim \phi$) if and only if there exists a sequence $B_1, \ldots, B_m$ such that $\phi = B_m$, and, for each $B_i \in \{B_1, \ldots, B_m\}$, either B_i is an axiom of $\mathcal{L}$, or B_i is in Φ, or B_i is a direct consequence of the preceding members of the sequence using modus ponens or instantiation of a universally quantified sentence. This argument definition is well-understood in the AI literature [4,23].

Argumentation. While an argument can be constructed by combining explicitly expressed knowledge (e.g., from a knowledge base), the aim here is to start from a conclusion and build arguments that support it from the knowledge that stakeholders provide and that can be related to the conclusion. Argumentation of a conclusion R consists of recursively defining an argument tree AT_R as follows:

1. Define R as the root of the tree AT_R and set $c = R$;
2. Let $\langle P, c \rangle$. Identify $p_1, \ldots, p_n$ s.t. $\{p_1, \ldots, p_n\} = P$, $P \subseteq K \cup \Delta^{\downarrow}$;
3. Define a node for each premise $p_i \in P$ and define an edge from that node to c. Draw the edge "$\longrightarrow$" if $p \in K$, or "$\longmapsto$" in case $p \in \Delta^{\downarrow}$;
4. Set $c = p_i$ and repeat steps 2 and 3 for each $i = 1, \ldots, n$, until the argument tree has been constructed to a satisfactory extent.

Argument Relationships. Of particular interest in argumentation is to confront arguments and reject some conclusion in favor of other. It is therefore necessary to define several simple relationships between arguments.

Two arguments $\langle P_1, c_1 \rangle$ and $\langle P_2, c_2 \rangle$ *disagree*, denoted by $\langle P_1, c_1 \rangle \bowtie_{\mathcal{K}} \langle P_2, c_2 \rangle$, if and only if $\mathcal{K} \cup \{c_1, c_2\} \vdash \bot$.

Instead of seeking contradiction of conclusions, a *counterargument* relation looks for incompatibility of a conclusion with the conclusion of a subargument of another argument. $\langle P_1, c_1 \rangle$ *counterargues at* c the argument $\langle P_2, c_2 \rangle$, denoted by $\langle P_1, c_1 \rangle \not\hookrightarrow^c \langle P_2, c_2 \rangle$, if and only if there is a subargument $\langle P, c \rangle$ of $\langle P_2, c_2 \rangle$ such that $\langle P_2, c_2 \rangle \bowtie_{\mathcal{K}} \langle P, c \rangle$ (i.e., $\langle P, c \rangle \subset \langle P_2, c_2 \rangle$ and $\mathcal{K} \cup \{c_1, c\} \vdash \bot$).

In case two arguments are such that one counterargues the other, it is necessary to determine which of the two is to be maintained. An argument $\langle P_1, c_1 \rangle$ *defeats at* c an argument $\langle P_2, c_2 \rangle$, denoted by $\langle P_1, c_1 \rangle \gg^c \langle P_2, c_2 \rangle$, if and only if there is a subargument $\langle P, c\rangle$ of $\langle P_1, c_1\rangle$ such that (1) $\langle P_1, c_1\rangle \not\leftrightarrow^c \langle P_2, c_2\rangle$ (that is, $\langle P_1, c_1\rangle$ counterargues $\langle P_2, c_2\rangle$ at c); and (2) $\langle P_1, c_1\rangle \succ^c \langle P, c\rangle$ ($\langle P_1, c_1\rangle$ is more *specific* than $\langle P, c\rangle$). In a dialectical tree (see below), defeat is represented by "$\not\longrightarrow$" directed from the conclusion of the argument that defeats to the node which is defeated. The specificity relation "$\succ^c$" is an order relation over arguments, defined so that arguments containing more information, i.e., which are more specific, are preferred over other. An argument $\langle P_1, c_1\rangle$ is *strictly more specific than* $\langle P_2, c_2\rangle$, denoted by $\langle P_1, c_1\rangle \succ^c \langle P_2, c_2\rangle$ if and only if (1) $\forall e \in \mathcal{K}_C$ such that $\mathcal{K}_N \cup \{e\} \cup P_1 \mid\!\sim c_1$ and $\mathcal{K}_N \cup \{e\} \not\mid\!\sim c_1$, also $\mathcal{K}_N \cup \{e\} \cup P_2 \mid\!\sim c_2$; and (2) $\exists e \in \mathcal{K}_C$ such that: (2.1) $\mathcal{K}_N \cup \{e\} \cup P_2 \mid\!\sim c_2$; (2.2) $\mathcal{K}_N \cup \{e\} \cup P_1 \not\mid\!\sim c_1$; (2.3) $\mathcal{K}_N \cup \{e\} \not\vdash c_2$.

Justification. Argument defeat is employed when attempting to justify a particular conclusion. The *justification* process consists of recursively defining and labeling a *dialectical tree* $\mathcal{T}\langle P, c\rangle$ as follows:

1. A single node containing the argument $\langle P, c\rangle$ with no defeaters is by itself a dialectical tree for $\langle P, c\rangle$. This node is also the root of the tree.
2. Suppose that $\langle P_1, c_1\rangle, \ldots, \langle P_n, c_n\rangle$ each defeats $\langle P, c\rangle$. Then the dialectical tree $\mathcal{T}\langle P, c\rangle$ for $\langle P, c\rangle$ is built by placing $\langle P, c\rangle$ at the root of the tree and by making this node the parent node of roots of dialectical trees rooted respectively in $\langle P_1, c_1\rangle, \ldots, \langle P_n, c_n\rangle$.
3. When the tree has been constructed to a satisfactory extent by recursive application of steps 1 and 2 above, label the leaves of the tree *undefeated* (U). For any inner node, label it undefeated if and only if every child of that node is a *defeated* (D) node. An inner node will be a defeated node if and only if it has at least one U node as a child. Do step 4 below after the entire dialectical tree is labeled.
4. $\langle P, c\rangle$ is a *justification* (or, P justifies c) if and only if the node $\langle P, c\rangle$ is labeled U.

Dialectical trees are shown in the UML-TAM traceability templates in Figures 4 and 5, in §6; arguments are drawn enclosed in boxes, a dialectical tree relates such boxes with the defeat relationship. The content of arguments is informally expressed, and can be replaced (pending some adjustments) with first-order formulae. However, the informal character thereof does not affect the ability to manually determine relationships between arguments, as they have been presented above, and consequently to proceed to justification. Having formal foundations, as suggested in the present subsection contributes to the precision of the conceptual bases for the argumentation and justification activities.

5.3 UML-TAM Connectors

Connectors in UML-TAM relate information used and produced with the design rationale, and argumentation and justification techniques to the content of the UML diagrams whose rationale traceability is to be ensured. Fig.3 shows the metamodel, written in UML class diagram notation, integrating the relevant concepts of UML-TAM and relating them to the UML 2.0 metamodel [18]

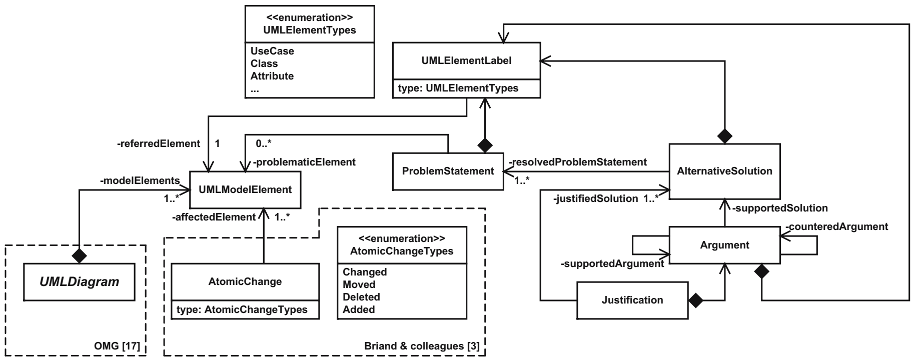


Fig. 3. Metamodel relating UML-TAM to the UML 2.0 metamodel

through the *UMLDiagram* class. Although the illustration §6 discusses the traceability in use case and class diagrams, the metamodel does not limit the potential for bridging UML-TAM and other UML diagrams.

The part of the metamodel proper to UML-TAM integrates the concept of ProblemStatement, AlternativeSolution, Argument, and Justification, each following the definitions given in previous subsections. Note the ProblemStatement can be associated to no UMLModelElement, which occurs when the ProblemStatement results in the addition of new UMLModelElement instances into a UMLDiagram instance. The metamodel is linked to a part of the metamodel underlying the bi-directional link traceability approach from Briand and colleagues [3]: AtomicChange is a modification applicable to the UML diagram, whose execution gives rise to a number of traceability links to ensure that information about what changed and how is captured. The types of atomic changes given in the figure are the basic ones, whereby more extensive taxonomies are suggested by refining each of the four activities, and this depending on the syntax of the underlying UML diagram [3]. An important practical consequence of the above metamodel is that UML-TAM can be thus be combined to automated traceability methods and applied selectively, when stakeholders explicitly identify problems which in turn entail the use of UML-TAM for resolution.

As the content of arguments can be informal or formal, labels are used to highlight the relevant elements of the UML model being mentioned in arguments, alternative solutions, and/or problem statement. The UMLElementLabel concept is thus introduced in the metamodel in Fig.3. In Fig.4, labels are placed within arguments and the alternative solution, whereas the problem statement (the title of the UML-TAM traceability template) does not contain explicit references to elements of the use case diagram, and therefore contains no labels.

The approach to relate the UML artifacts and those produced in UML-TAM is straightforward: as soon as a justified alternative solution is found, and the stakeholders no longer provide arguments to defeat it (i.e., the justification process ends), change is performed in the corresponding UML diagram. A template is filled out—it contains a snapshot showing the original structure of the

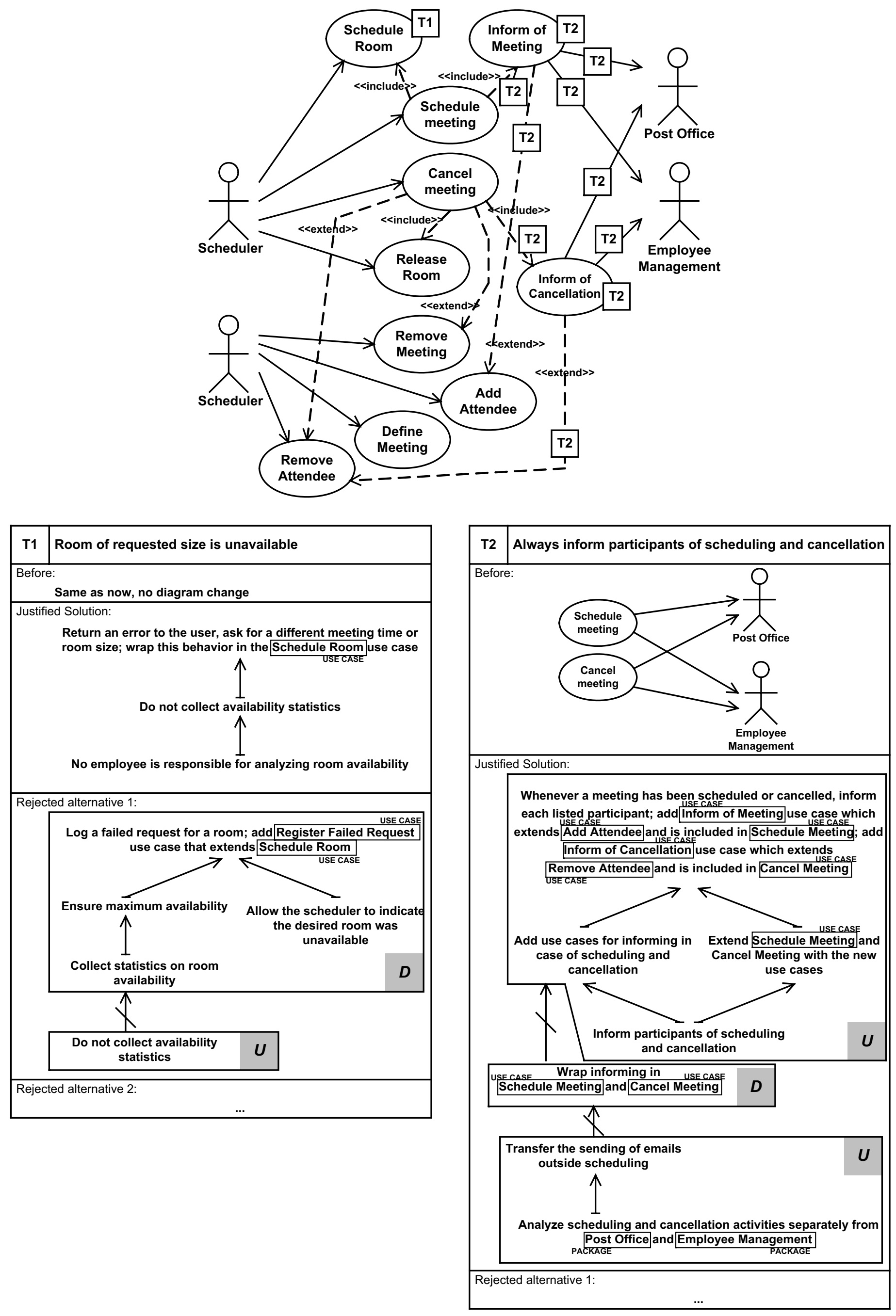

Schedule Room
T1
Inform of Meeting
T2
<<include>>
Schedule meeting
Post Office
Cancel meeting
<<include>>
<<extend>>
Scheduler
Release Room
Employee Management
Inform of Cancellation
<<extend>>
Remove Meeting
<<extend>>
Add Attendee
Define Meeting
Remove Attendee
T1 Room of requested size is unavailable
Before:
Same as now, no diagram change
Justified Solution:
Return an error to the user, ask for a different meeting time or room size; wrap this behavior in the Schedule Room use case
USE CASE
Do not collect availability statistics
No employee is responsible for analyzing room availability
Rejected alternative 1:
Log a failed request for a room; add Register Failed Request use case that extends Schedule Room
Ensure maximum availability
Allow the scheduler to indicate the desired room was unavailable
Collect statistics on room availability
D
Do not collect availability statistics
U
Rejected alternative 2:
...
T2 Always inform participants of scheduling and cancellation
Before:
Schedule meeting
Cancel meeting
Post Office
Employee Management
Justified Solution:
Whenever a meeting has been scheduled or cancelled, inform each listed participant; add Inform of Meeting use case which extends Add Attendee and is included in Schedule Meeting; add Inform of Cancellation use case which extends Remove Attendee and is included in Cancel Meeting
Add use cases for informing in case of scheduling and cancellation
Extend Schedule Meeting and Cancel Meeting with the new use cases
Inform participants of scheduling and cancellation
U
Wrap informing in Schedule Meeting and Cancel Meeting
D
Transfer the sending of emails outside scheduling
U
Analyze scheduling and cancellation activities separately from Post Office and Employee Management
PACKAGE
Rejected alternative 1:
...

Fig. 4. The modified use case diagram with accompanying rationale traceability information produced with UML-TAM

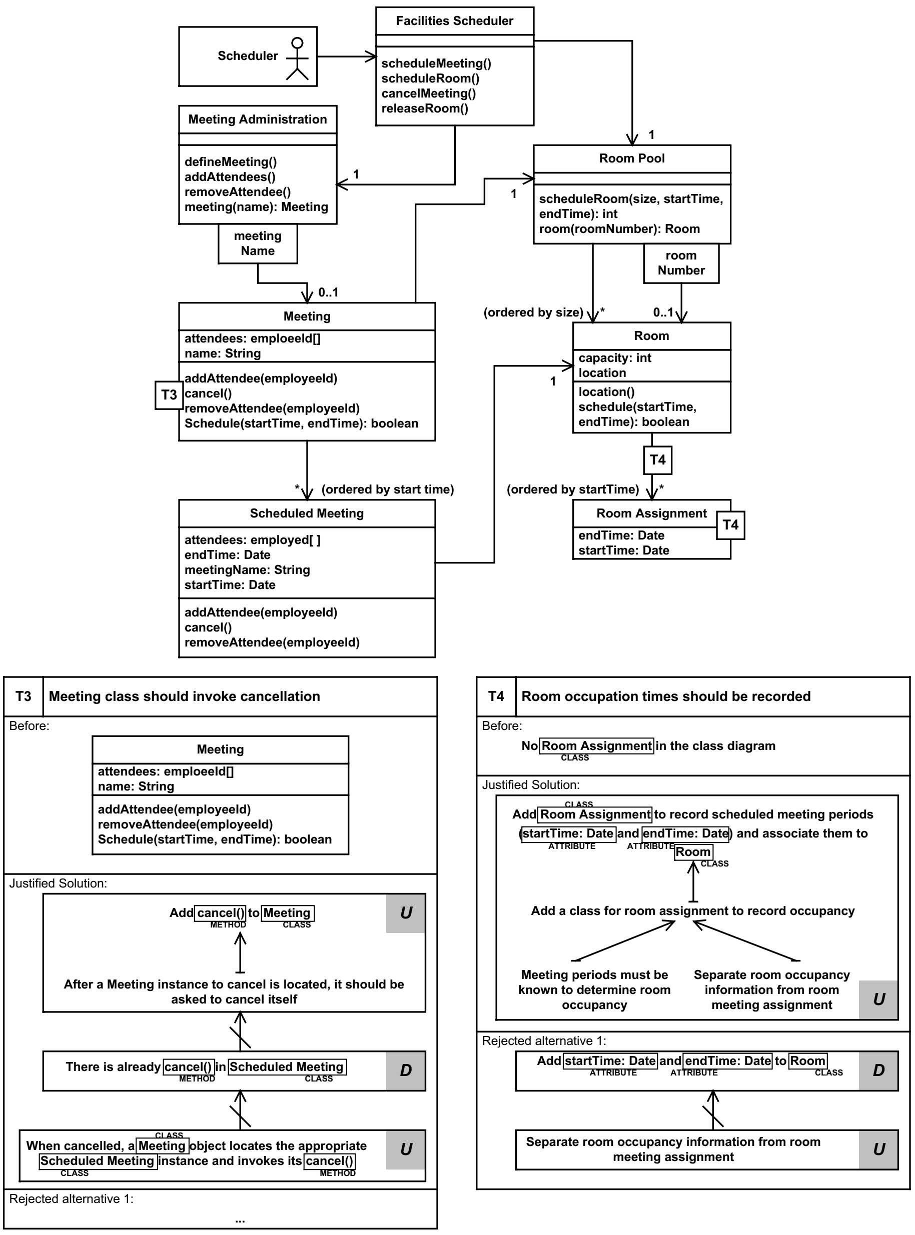

Scheduler
Facilities Scheduler
scheduleMeeting()
scheduleRoom()
cancelMeeting()
releaseRoom()
Meeting Administration
defineMeeting()
addAttendees()
removeAttendee()
meeting(name): Meeting
meeting Name
Room Pool
scheduleRoom(size, startTime, endTime): int
room(roomNumber): Room
room Number
1
0..1
(ordered by size)
*
Meeting
attendees: emploeeId[]
name: String
addAttendee(employeeId)
cancel()
removeAttendee(employeeId)
Schedule(startTime, endTime): boolean
T3
Room
capacity: int
location
location()
schedule(startTime, endTime): boolean
T4
(ordered by start time)
(ordered by startTime)
Scheduled Meeting
attendees: employed[ ]
endTime: Date
meetingName: String
startTime: Date
addAttendee(employeeId)
cancel()
removeAttendee(employeeId)
Room Assignment
endTime: Date
startTime: Date
T3 Meeting class should invoke cancellation
Before:
Meeting
attendees: emploeeId[]
name: String
addAttendee(employeeId)
removeAttendee(employeeId)
Schedule(startTime, endTime): boolean
Justified Solution:
Add cancel() METHOD to Meeting CLASS
U
After a Meeting instance to cancel is located, it should be asked to cancel itself
There is already cancel() METHOD in Scheduled Meeting CLASS
D
When cancelled, a Meeting CLASS object locates the appropriate Scheduled Meeting CLASS instance and invokes its cancel() METHOD
U
Rejected alternative 1:
...
T4 Room occupation times should be recorded
Before:
No Room Assignment CLASS in the class diagram
Justified Solution:
Add Room Assignment CLASS to record scheduled meeting periods (startTime: Date ATTRIBUTE and endTime: Date ATTRIBUTE) and associate them to Room CLASS
Add a class for room assignment to record occupancy
Meeting periods must be known to determine room occupancy
Separate room occupancy information from room meeting assignment
U
Rejected alternative 1:
Add startTime: Date ATTRIBUTE and endTime: Date ATTRIBUTE to Room CLASS
D
Separate room occupancy information from room meeting assignment
U

Fig. 5. The modified class diagram with accompanying rationale traceability information produced with UML-TAM

part of the diagram that is being changed, the problem statement, the alternative solutions, the justification, and all arguments provided for each alternative solution.

6 Applying UML-TAM

It has been observed earlier that the initial use case diagram shown in Fig.1(a) is incomplete in several respects. Using UML-TAM, two changes were performed leading to the use case diagram in Fig.4. There, labels are placed on the elements of the diagram to relate them to *traceability templates* used in UML-TAM to summarize information used and produced in moving from the initial version of the diagram to that presented in Fig.4. Each template contains four parts: (i) a label (e.g., T1, T2) for relating the elements of the diagram to the template; (ii) a title, which is the problem statement requiring diagram change; (iii) the dialectical tree for the justified alternative solution; and (iv) the dialectical trees for the rejected alternative solutions. Following the metamodel in Fig.3, information referring to UML diagram elements and appearing in the template is labeled following the kind of UML element the information refers to.

Figures 4 and 5 are self-explanatory and show modified initial use case and class diagrams obtained by applying UML-TAM. Each has been constructed by applying the UML-TAM. Practical experience with UML-TAM that goes beyond the simple, yet illustrative case presented here leads to several observations about the practical use of the proposed method. For instance, it has been empirically observed that nonmonotonic reasoning is hard for humans [8]. Effort involved in finding arguments in UML-TAM is considerable and appears to confirm the cited empirical result. Some techniques derived from theory are particularly hard to apply in practice: for instance, comparing arguments for specificity appeared counterintuitive and was thus seldom used. Prior experience and resources about the debated domain are relevant sources of arguments, so that referring to these is suggested. Although the difficulties are considerable when applying argumentation and justification, a significant benefit is that these techniques lead to the externalization of information usually left implicit in UML modeling. The information made explicit is available to a number of stakeholders who can, through argumentation and justification, question and revise the modeling decisions. Moreover, lessons can be learned from past modeling problems as sources of the problems (such as, e.g., fallacious argumentation) can be identified by going back to the recorded arguments. UML-TAM is therefore of interest for projects in which particularly high degree of rigor is required, as in the case of, e.g., safety-critical systems.

7 Conclusion and Future Work

The UML Traceability through Argumentation Method presented herein introduces rigorous argumentation and justification when tracing the rationale behind UML modeling decisions. The main contributions are: (1) The information about the design rationale used in modeling is usually lost or, when available, stated in an unstructured manner. UML-TAM provides a simple, yet precise means for representing this information, analyzing it for problematic rationale (by justification), and using it to arrive at justifiably appropriate modeling decisions. (2) Both qualitative and quantitative, informal and formal information can be put into arguments allowing the application of the method to a wide range of settings. (3) When combined with traceability approaches focused on answering how, what, when, and who modified a UML diagram, UML-TAM allows answering and discussing *why* a change was needed. (4) By applying argumentation and justification activities, the modeler can claim that a modeling choice is appropriate or not, while relying on solid and well understood conceptual foundations and rigorous processes for their use. Modeling choices can thus be claimed as justified, or questioned through a step-by-step process. Following the outline of related research efforts §2, the proposed method advances the rationale traceability literature, while ensuring compatibility with approaches focusing on traceability of other types of information—this is accomplished by focusing the method on a precise traceability issue, proposing connection points for relating the method to compatible approaches, and avoiding overlap with related techniques.

Current effort includes the exploration of benefits of formalizing arguments in combination with various UML formalizations, to attempt automated analysis of argument and associated UML diagram structures. Experimentation is currently performed to improve usability in industrial settings.

Acknowledgments. The authors thank Alex Borgida for insightful remarks on the paper [14] presented at the 14th International Conference on Requirements Engineering, which helped shape the ideas herein. The first author acknowledges funding from the Belgian ICM/CIM Doctoral Fellowship Program.

References

1. Antoniol, G., Canfora, G., De Lucia, A.: Maintaining traceability during object-oriented software evolution: a case study. In: Proc. Int. Conf. Softw. Maintenance (1999)
2. Besnard, P., Hunter, A.: A logic-based theory of deductive arguments. Intell. 128(1–2), 203–235 (2001)
3. Briand, L.C., Labiche, Y., Yue, T.: Automated Traceability Analysis for UML Model Refinements. Carleton Univ. Technical Report, TR SCE-06-06, ver.2 (August 2006)
4. Chesñevar, C.I., Maguitman, A.G., Loui, R.P.: Logical Models of Argument. ACM Comput. Surv. 32(4), 337–383 (2000)
5. Conklin, J., Begeman, M.L.: gIBIS: A hypertext tool for exploratory policy discussion. ACM Trans. Inf. Syst., 6(4) (1988)
6. Dömges, R., Pohl, K.: Adapting Traceability Environments to Project-Specific Needs. Comm. ACM 41(12), 54–62 (1998)
7. Egyed, A.: A Scenario-Driven Approach to Traceability. Proc. Int. Conf. Softw. Eng., 123–132 (2001)
8. Ford, M., Billington, D.: Strategies in Human Nonmonotonic Reasoning. Computat. Intel. 16(3), 446–468 (2000)
9. Gotel, O.C.Z., Finkelstein, A.C.W.: An Analysis the Requirements Traceability Problem. Tech. Rep. TR-93-41, Dept. of Computing, Imperial College (1993)
10. Gotel, O.C.Z., Finkelstein, A.C.W.: An analysis of the requirements traceability problem. In: Proc. Int. Conf. Req. Eng., pp. 94–101 (1994)

11. Gotel, O.C.Z.: Contribution Structures for Requirements Engineering. Ph.D. Thesis, Imperial College of Science, Technology, and Medicine, London, England (1996)
12. Haumer, P., Pohl, K., Weidenhaupt, K., Jarke, M.: Improving Reviews by Extending Traceability. In: Proc. Annual Hawaii Int. Conf. on System Sciences (1999)
13. Jackson, J.: A Keyphrase Based Traceability Scheme. IEE Colloq. on Tools and Techn. for Maintaining Traceability During Design (1991)
14. Jureta, I.J., Faulkner, S., Schobbens, P.-Y.: Justifying Goal Models. Proc. Int. Conf. Req. Eng., 119–128 (2006)
15. Letelier, P.: A Framework for Requirements Traceability in UML-Based Projects. In: Proc. Int. Worksh. on Traceability in Emerging Forms of Softw. Eng. (2002)
16. Louridas, P., Loucopoulos, P.: A Generic Model for Reflective Design. ACM Trans. Softw. Eng. Meth. 9(2) (2000)
17. Naslavsky, L., Alspaugh, T.A., Richardson, D.J., Ziv, H.: Using Scenarios to Support Traceability. Proc. Int. Worksh. on Traceability in emerging forms of software engineering, 25–30 (2005)
18. OMG. UML 2.0 Superstructure Specification. Object Management Group, Final Adopted Specification ptc/03-08-02 (2003)
19. Pinheiro, F.A.C., Goguen, J.A.: An Object-Oriented Tool for Tracing Requirements. IEEE Software 13(2), 52–64 (1996)
20. Pohl, K.: Process-Centered Requirements Engineering. Advanced Software Development Series. J.Wiley & Sons Ltd, Taunton, England (1996)
21. Pohl, K.: PRO-ART: Enabling Requirements Pre-Traceability. Proc. Int. Conf. Req. Eng., 76–85 (1996)
22. Pohl, K., Dömges, R., Jarke, M.: Towards Method-Driven Trace Capture. Proc. Conf. Adv. Info. Syst. Eng., 103–116 (1997)
23. Prakken, H., Vreeswijk, G.: Logical systems for defeasible argumentation. In: Gabbay, D., Guenther, F. (eds.) Handbook of Philosophical Logic, Kluwer, Dordrecht (2002)
24. Ramesh, B., Dhar, V.: Supporting systems development by capturing deliberations during requirements engineering. IEEE Trans. Softw. Eng. 18(6), 498–510 (1992)
25. Ramesh, B., Stubbs, C., Powers, T., Edwards, M.: Implementing requirements traceability: A case study. Annals of Softw. Eng. 3, 397–415 (1997)
26. Simari, G.R., Loui, R.P.: A mathematical treatment of defeasible reasoning and its implementation. Artificial Intelligence 53, 125–157 (1992)
27. Toranzo, M., Castro, J.: A Comprehensive Traceability Model to Support the Design of Interactive Systems. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 283–284. Springer, Heidelberg (1999)
28. Ubayashi, N., Tamai, T., Sano, S., Maeno, Y., Murakami, S.: Model evolution with aspect-oriented mechanisms. In: Proc. Int. Worksh. Principles of Softw. Evol. (2005)
29. van Lamsweerde, A., Darimont, R.: Massonet Ph.: The Meeting Scheduler Problem: Preliminary Definition. Université catholique de Louvain (1992)
30. van Lamsweerde, A.: Goal-Oriented Requirements Engineering: A Guided Tour. In: Proc. Int. Conf. Req, pp. 249–263 (2001)